Weekend Fitness Centre

Assignment 2, Coursework

Table of Contents

Introduction

The goal of this project is to construct a self-contained booking system using the Java programming language for a fitness centre that operates at weekends. Important information to track includes members, classes, exercises (types of class) and bookings. The relationships within the data will be significant too, for instance, bookings will manage the many-many relationship between members and classes.

The system should be easy to use and maintain, and meet the specification of the project.

Requirements

The club needs to be able to manage bookings (Bookings) made by customers (Members). Each booking relates to a particular class (Class), which can be of a particular kind (Exercise). Therefore the main component classes in the program shall be Bookings, Members, Classes and Exercises.

The system will need to be able to add new examples of each, on demand, which will be stored within the program. There is to be no data held externally to the program, which will have either a CLI (Command Line Interface) or GUI (Graphical User Interface).

Reports

The user of the system should be able to obtain reports indicating:

> The number of members booked into each exercise, and the average rating.
>
> The best performing exercise, which has achieved the most income.

This should be searchable by month.

Bookings

Members must book into specific class times – therefore an individual booking can contain only one member and one class. The payment method (card or cash) must also be recorded.

A booking can be changed to allow the member to book into an alternative class instead, as long as there is space available. After the class date, a review and rating (One to Five) can be left by the customer.

Members should not be prevented from booking as many classes as they wish. Creating a booking should automatically reduce the spaces available, and increase the takings of the associated class. Likewise, cancellation of the booking should have the opposite effect.

Classes

The club can plan up to 4 classes on any Saturday or Sunday, for up to 20 people each. The available slots for each class are Morning, Afternoon, Evening 1 and Evening 2. Classes cannot be booked into a slot and date if that slot and date is already allocated to another class.

Exercises

The price is unique to the type of the class, therefore the Exercise will contain the class price.

Test Data

The program should contain some pre-loaded test data including at a minimum:

5 types of classes (Exercises), 15 customers (Members), 20 reviews (with rating)

Assumptions

General

I have assumed that the fitness club will have an external system for ensuring GDPR compliance and managing secure access to their IT systems and data.

Reports

These are searchable by Month, but this would be rendered meaningless if the year was not also checked, therefore this should also be available as an option.

Members

We are given no information around which to base our construction of this class around, therefore it is reasonable to assume that this object should have first and last name fields, and that the unique identifier of this class should not relate to these fields, as it is not uncommon for people to have the same names.

Cancellation of a membership should cause the cancellation of all associated bookings.

Bookings

It is reasonable to assume that a member should not be able to book the same class twice. If they wish to bring a guest, this will be possible by the creation of a new member.

Only one rating and rating should be possible per booking, which can be added to or amended at a later. If this were not the case, then a user could disproportionately affect the average rating by continually leaving new ratings. Therefore, the rating and review properties will be built into the Booking class. The absence of a rating should not imply a zero rating, therefore it makes sense to add ‘None’ as a default option for this. When reporting on ratings, if all bookings have ‘None’ as their rating, this should be reported as ‘Not yet rated’, or something similar.

A review should not be made unless a class has actually been attended, in order to ensure that the review is genuine. Therefore the booking class will contain a ‘Status’ property to allow this to be recorded and checked.

Cancellation of a booking is discussed in the requirements.

Classes

New classes can be added to the system, but it should not be possible to plan new classes in the past.

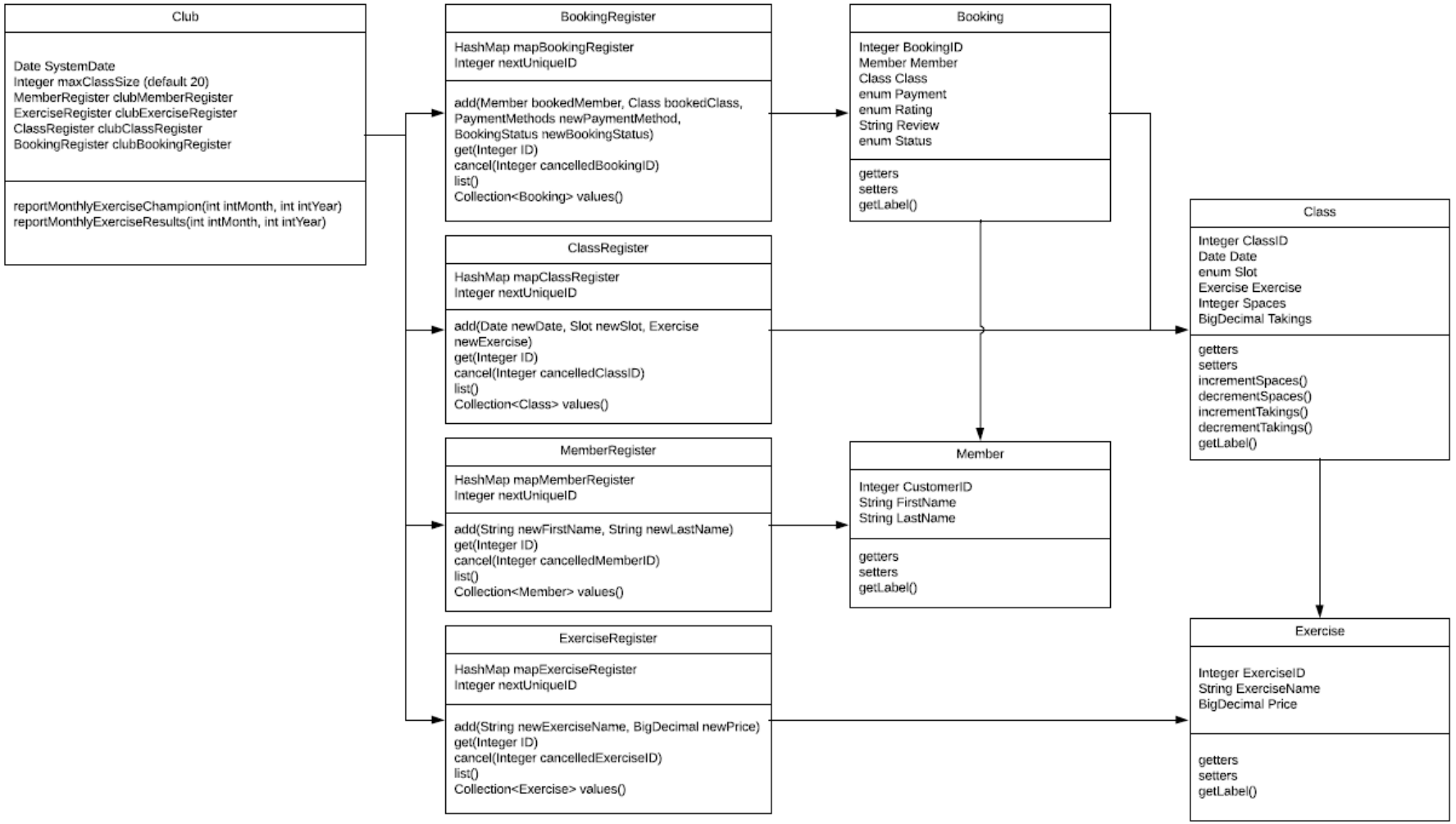
Cancellation of a class should cause the cancellation of all associated bookings.

Exercises

Cancellation of an exercise should cause the cancellation of all associated classes and bookings.

Structure

Class diagram



Class description

The diagram above shows the main classes used in the back end of the program, to manage the bookings and related information. I have broken the program down into ‘Register’ classes (with Register in their name), and ‘Registerable’ classes (e.g. Booking, Exercise etc.), which are held within them. The idea of this is to provide the Club class with a means of cataloguing, editing, and retrieving the individual data objects as required.

In addition, some classes are partially comprised of other classes. For instance, Booking has an attribute of Member and of Class, along with attributes entirely of it’s own.

Each ‘Register’ classes is provided with a HashMap for storing and retrieving the relevant ‘Registerable’ class, and a nextUniqueID attribute, which is used for assigning unique identifiers within the HashMap. There are also a set of common methods, including add(), get(), cancel(), list() and values(). These are used, respectively, for adding new objects, retrieving them using their unique ID, cancelling (removing) them, listing them for testing purposes and returning them as a collection for other classes to iterate through. In particular, the cancellation method is often required to cause cancellations in other objects through the relevant method calls, for instance, when a membership is cancelled, the associated bookings should be cancelled.

‘Registerable’ classes on the other hand, have data and methods much more closely linked to the individual records they represent, including dates, names and selections (implemented using enums discussed later). Similarly, they contain getters(accessors) and setters(mutators) for each attribute. In particular, Class contains setters for incrementing and decrementing the Takings (by the value of the exercise price) and Spaces. This is important, as these properties are never directly set once initialised, instead they are modified incrementally or decrementally as bookings are made or cancelled.

Data Structures

The system has made use of a variety of data types. At the ‘Register’ level, HashMaps are used to ensure that each object is stored together with a unique identifier in order to make later retrieval or removal easy. ArrayLists are also used, but more commonly in situations where a simple iteration function is required.

As the program called for the tracking of pricing information, the use of floats was not going to be acceptable. For this reason, pricing has been implemented using the BigDecimal data type. This ensures accuracy while also supporting decimal points and enforced scaling, which makes it ideal for pricing purposes.

I have tended to prefer Integer over int, as the former provides a wider variety of functionality, for instance conversion to strings and compatibility with HashMap keys (as Integer is a reference type).

The Calendar type has also been useful for providing enhanced Date functionality, for instance checking the day of week. A variety of graphical data types has been used as appropriate, ComboBox for selections, text fields for entering strings, etc.

Design

In general, I have designed the system to seperate back end and front end functionality. In particular, this means that the Club object handles the data and the GUIWeekendFintessclass object handles the user interface. This makes the code more easy to read, understand and maintain as it is easier to locate bugs. More importantly, the validation of data being entered occurs only in one place. This makes it much harder for bugs to emerge because future developers do not have to remember when they do or don't need to implement additional validation.

I have attempted to maintain a consistent approach to meaningful variable names. This is important for debugging purposes as it becomes easier to identify which variables hold what kind of data and for what purpose. To this end I have followed a convention of including the data type, related class, and purpose of the variable in it's name.

The Register classes are actually inner classes, held within the Club class. Implementation as classes is not strictly necessary, however, this provides encapsulation of the individual HashMaps, increases readability, and makes the code more easily maintainable. Additionally, the selection of inner classes means that they cannot be instantiated separately from the Club, which would be meaningless. Register classes validate data at the level of the HashMap, for instance, checking that the class is not being booked for a time already taken by another class.

Registerable classes are external to the Club class, and contains a second level of validation specific to individual items, eg. checking that the intended date is not in the past, when a class is created. This item level validation provides a greater level of consistency as it means that validation occurs automatically in all the places where the class is instantiated.

One particularly interesting method of the Register classes, is the cancel() method. This is because the cancellation of some objects needs to bring about the cancellation of some other objects. It was important to implement this only once, to ensure consistency. Otherwise the data would very quickly unravel. In essence, these cancellation methods work by retrieving related objects and checking if they contain the object being cancelled. If this is the case, they can then be cancelled themselves.

On the interface side, I have designed the event handlers with seperate functions to implement their functionality. This means that handlers can share functionality when required, for instance when clicking the 'save' button, after making changes, the program then calls the appropriate 'find' function – to return the user to the find screen. This ensures consistency, because code is not being repeated in different locations.

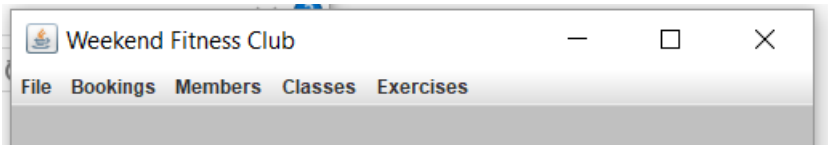
Some options required within the program are limited to a small variety of selections that does not change over the life time of the program and must be entered correctly, eg. time slots (Slot), payment methods (PaymentMethods) and booking status (BookingStatus). Therefore these have been implemented using enums.

Testing

The program provides a consistent user interface for all but the ‘File’ menu item. Therefore I have provided a run through of testing for the ‘Bookings’ menu item, other menu items have been similarly tested and can be demonstrated using the same process.

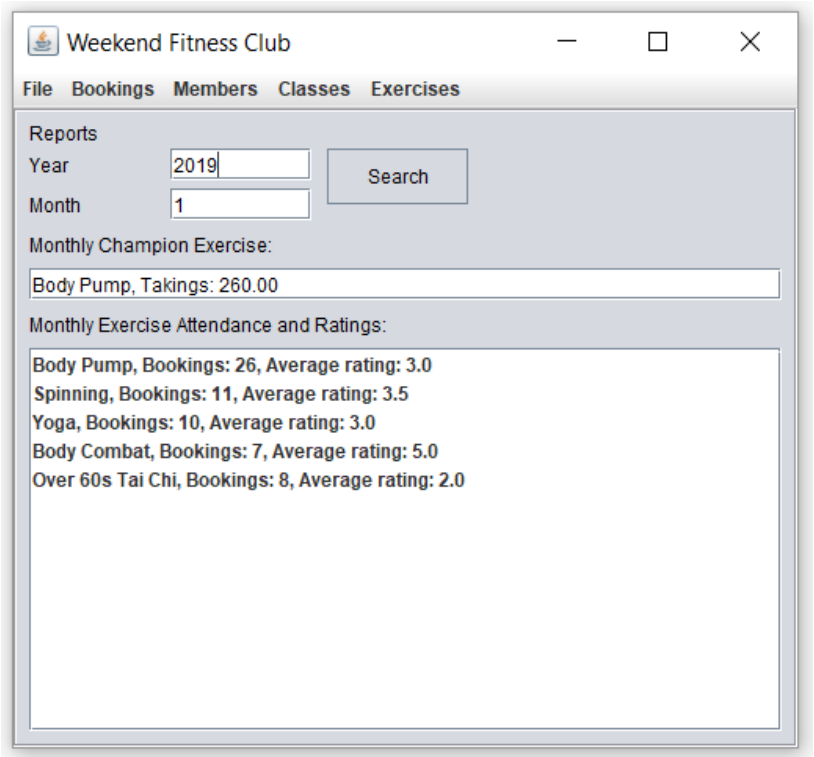
Menu items

On first opening the program, the user is faced with the following menu options, which will remain visible for the duration of operation.



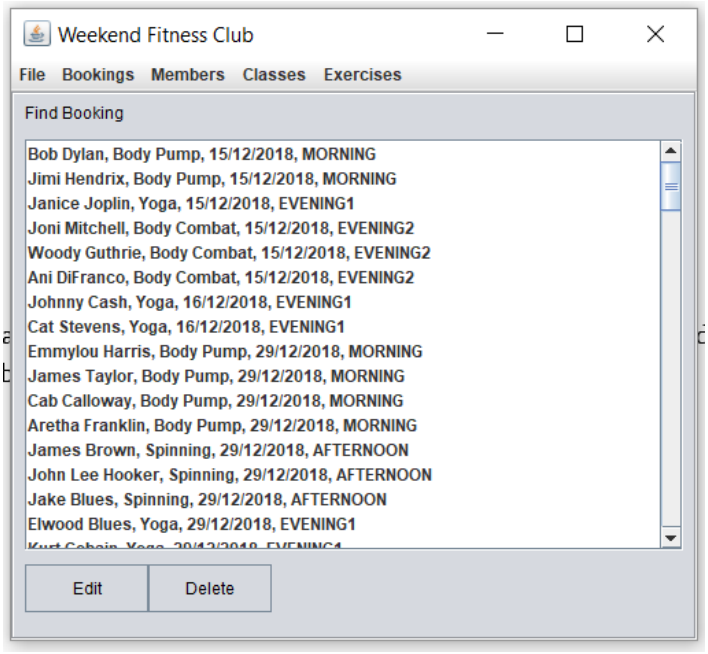
Reports

Available from the ‘File’ menu item, this provides the report information for the selected month (1-12 for January to December respectively) and year. By clicking the search button, the results are displayed. The current year and month are chosen by default.



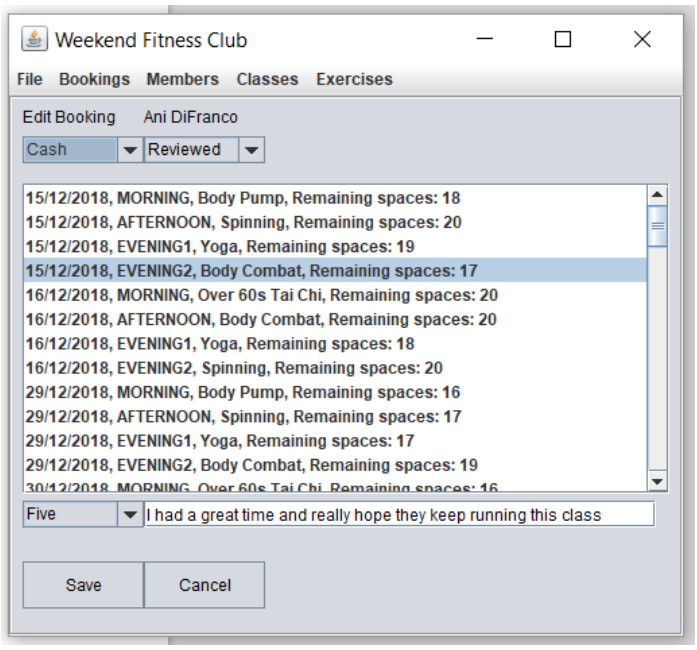
Find

By clicking Bookings > Find, the user can see the bookings currently available in the system, in order of entry. From here, the user can edit or delete a booking by selecting from the list and clicking the appropriate button, the results of which is explored separately below.



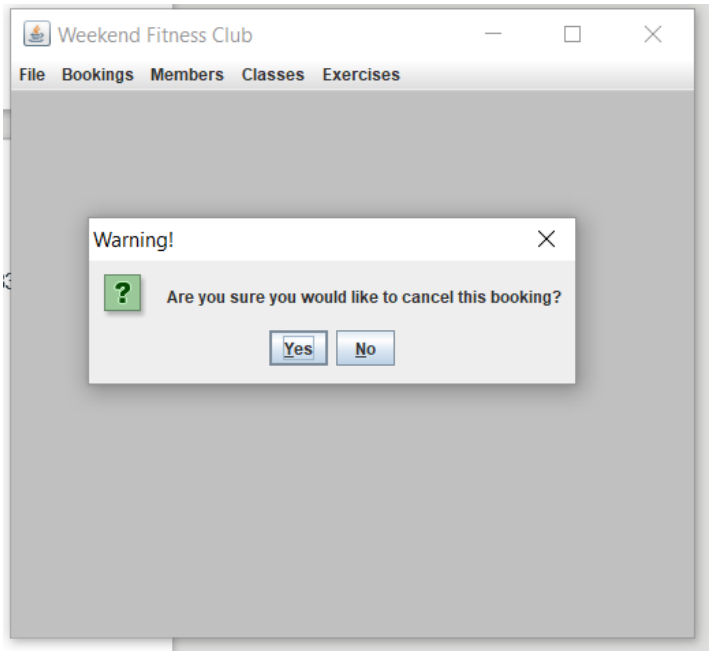
Edit

The edit button will present the user with the options available for the booking. This is committed to memory by using the save button, which will return the user to the ‘Find’ screen if successful. If an invalid selection or entry is made, the user is informed via pop-up and returned to the ‘Edit’ screen. Likewise, the cancel button will return the user to the ‘Find’ screen without making any changes. Please note that when changing the class, this sends the booking to the bottom of the ‘Find’ screen.



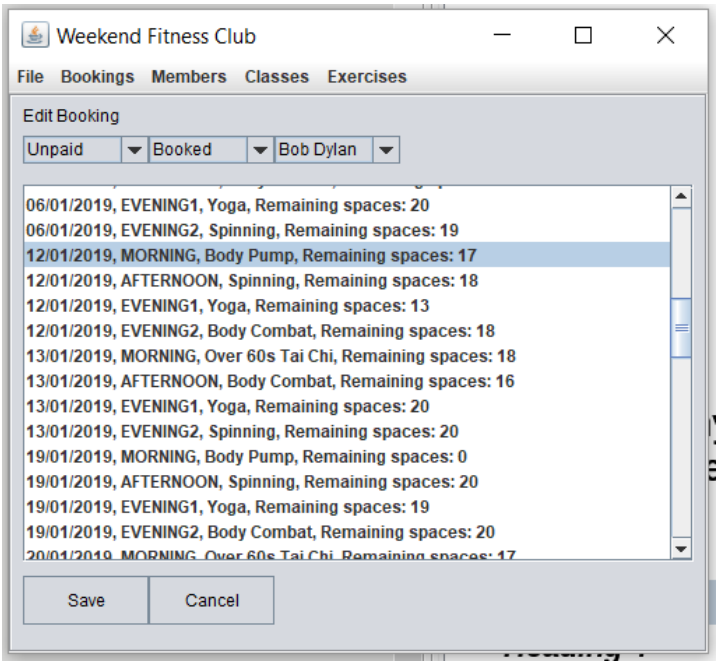
Delete

From the ‘Find’ screen, after selecting the item and clicking the delete button, the user will be prompted with a pop-up message to confirm their choice. After this, the user will be returned to the ‘Find’ screen, with the deleted booking removed if appropriate.



Add

By clicking Bookings > Add, the user is able to add new bookings to the system. This is very similar to the Edit screen and works in exactly the same way as it uses many of the same objects, attributes and methods. Simply make the appropriate selections and click Save.



Conclusions

Improvements

If I were to return to work on this project, there are some areas that I could improve. In particular, the project is currently very highly coupled, especially directly between the GUI class and the individual classes (Booking, Member etc.). This was not intended in my original design, and would be better implemented through additional accessor and mutator methods in the Club class. This would then completely separate the user interface and back end of the program, which could be useful if the company wanted to later open up the system to external inputs via an API or web app. Additionally, this would ensure that all validation is maintained on the back-end which will make it more easily maintainable.

I have striven to make the code easily readable and thus maintainable. However, I am conscious that a large portion of the GUI code (GUIWeekendFitnessClub) is given over to assigning properties to GUI objects. This could be largely abstracted away by defining my own default classes that extended the respective graphical classes. Then, instead of instantiating a Java object:

JLabel myLabel = new Jlabel()

I could call on my own version for example:

DefaultLabel myLabel = new DefaultLabel()

Which would have the properties pre-defined in the Class definitions.

Additionally, I would make greater use of abstraction in my GUI implementation. For instance, many of the methods, properties and objects of my JPanels were very similar if not the same, and could have benefited from an abstract class that itself extended JPanel.

Finally, the various implementations of Register inner classes (eg. BookingRegister, MemberRegister) would have benefited from implementation of appropriate sorting (by date, alphabetical as appropriate) in the values() method. This would cause entries in the JList GUI objects to be sorted which would make the application more user friendly in turn. This would be particularly beneficial when editing Bookings as this process essentially ‘moves’ the booking to the end of the list (see Testing).

Observations

Having a background in functional programming, it has been very enlightening to experience object oriented programming first-hand. It ensures a level of quality assurance and code maintainability that is difficult to consistently provide otherwise.

The numerous ways in which classes can be built has given me a great deal of pause for thought, and there is not necessarily one ideal solution.

However, the way that properties and methods can be encapsulated, and classes abstracted or inherited from has provided me with the tools to develop a versatile, yet maintainable and robust program.